Всичко за lambda, accumulate, let & let*

1. Функции и процедури като аргументи на други функции.

Нека първо видим как се предават функциите като аргументи на други функции. При писането на код на Scheme това е улеснено поради две основни причини: не указваме типове на данните, с които работим, а за интерпретаторите всички наши дефиниции – дали на променливи или на функции – са равноправни, на всяко име (било то променлива или функция) се съпоставя по един и същ начин *обект*. Този обект може да е число, стринг, процедура дори, Scheme категоризира всички като *обект*. Това означава, че когато искаме да използваме функция като аргумент на такава от по-висок ред, трябва просто да предадем името на функцията по същия начин, по който предаваме имена на променливи насам-натам. Ето как се реализира и изпълнява функция, която приема едноаргументната числова функция F и две числа и връща разликата F(b)-F(a).

```
(define (diff F a b) (- (F b) (F a))
(define (sq x) (* x x))

>(diff sq 2 3)
>5
```

Функцията F, която искаме да използваме в тази от по-висок ред, предаваме само по име, т.е. името на процедурния обект, който смята x^2 за някое x – ние сме го кръстили **sq**. Обръщам внимание върху думите „едноаргументна“ и „числова“. Никой няма да ни спре да направим грешна дефиниция, напр. `(and (F a b) (F a))`. Интерпретаторът ще я разчете като синтактически правилна и ще остави на нас да подадем подходящо F, което хем да изпълнява всички условия на задачата, хем неговите извиквания да имат смисъл и да връщат смислени резултати. Такова неправилно ползване на предадената функция F би било наша грешка и интерпретаторите няма да ни я покажат, докато не извикаме и изпълним написаната от нас функция `diff`. При изискванията на задачата написаната горе дефиниция е коректна и вярна. Ако ѝ подадем аргументи, изпълняващи условията, ще извика F коректно и ще работи с върнатия от нея резултат като с число, каквото е условието.

Това са най-важните неща, които трябва да запомни човек относно предаването на функции като аргументи, в следващите примери с функции от по-висок ред ще бъдат илюстрирани отново.

2. Функции от по-висок ред: sum, product, accumulate и filter-accumulate. Примери.

Тези четири „стандартни“ функции от по-висок ред решават задачи, чието решение изисква някакъв вид последователно обхождане на интервал от числа и извършване на някакво действие (или повече) на всяка стъпка. Характерна е и рекурсивна зависимост в условията на задачите, показана в долните примери.

Да се дефинира функция, която намира сумата на числата в даден (целочислен) интервал:

$$f: a,b \to a+(a+1)+\ldots+b \Leftrightarrow \mathrm{Sum}[a;b]=\begin{cases} 0, a > b \\ a + \mathrm{Sum}[a+1; b], a \leq b \end{cases}$$

Първото условие можеше да е „b при a=b”, но тази може да улови грешни входни данни, както и да послужи когато обхождането не е от едно цяло число на следващото. Нашата функция ще изглежда така:

```
(define (sum-ints a b)
      (if (> a b) 0
          (+ a (sum-ints (+ a 1) b))))
```

Просто, рекурсивно, работещо. Аналогична задача би била да се намери сумата на кубовете на всяко второ число в интервала [a;b]:

$$f: a,b \to a^3+(a+2)^3+\ldots+b^3 \Leftrightarrow \mathrm{Sum}[a;b]=\begin{cases} 0, a > b \\ a^3 + \mathrm{Sum}[a+2; b], a \leq b \end{cases}$$

```
(define (sum-cubes a b)
      (if (> a b) 0
          (+ (expt a 3) (sum-cubes (+ a 2) b))))
```

Приликите между двете функции са очевидни, а аналогични примери има безброй много. Разликите са две – начинът, по който се взима всеки път общия член на сумата (a или a^3), както и стъпката за рекурсивното извикване (следващото или по-следващото число). Тези две „правила“ можем да изнесем като отделни, малки функции, които да предаваме на обобщена функция за сума по интервал, позволявайки ни да решаваме различните задачи чрез различни извиквания на една и съща функция. Нека кръстим правилата term и next – за общ член на сумата и правило за итериране, и ги добавим като аргументи на обобщената функция за сума. Използвайки знанията за предаване на функции като аргументи, можем да напишем следната функция от по-висок ред:

```
(define (sum term a next b)
      (if (> a b) 0
          (+ (term a) (sum term (next a) next b))))
```

Тези два нови аргумента трябва да „влачим“ винаги при рекурсивното извикване на функцията. Рекурсията се реализира чрез разглеждане на подинтервал с изместено начало (next a). Връщайки се към поставените две задачи, можем да ги решим така:

```
(define (sum-ints a b)
      (define (id x) x)
      (define (1+ x) (+ x 1))
      (sum id a 1+ b))

(define (sum-cubes a b)
      (define (cube x) (expt x 3))
      (define (2+ x) (+ x 2))
      (sum cube a 2+ b))
```

Решаването на този тип задачи се свежда само до измисляне и дефиниране на тези две вътрешни функции, които да предадем на функцията от по-висок ред sum. Тъй като в нейната

дефиниция използваме (term a), значи функцията, която предаваме като аргумента term, трябва да приема един аргумент и да връща число (защото събираме числа). Същото важи и за функцията next. Тези изисквания са към нас и е наша отговорност да ги спазим, защото евентуалната грешка няма да се забележи преди изпълнение на програмата. Още нещо – във вложените дефиниции използвахме x като името на формалния параметър. В общия случай то няма значение, може да бъде дори a или b, освен в случаите, когато ни трябват точно тези a и b, с които извикваме функцията ни.

Подобна на предните задача би била за намиране на произведението на числата от някой интервал [a;b]. Да, има различни начини за решение, но нека използваме същия подход, както за сумата:

f: a,b -> a*(a+1)*...*b $\Leftrightarrow$ $\text{Product}[a;b]=\begin{cases} 1, a > b \\ a * \text{Product}[a+1; b], a \leq b \end{cases}$

```
(define (product-ints a b)
      (if (> a b) 1
          (* a (product-ints (+ a 1) b))))
```

По същите съображения, както при функцията за сума по интервал, можем да напишем и обобщена функция за произведение:

```
(define (product term a next b)
      (if (> a b) 1
          (* (term a) (product term (next a) next b))))
```

Тук виждаме две дребни разлики с функцията sum – операцията, чрез която изчисляваме резултата, е умножение, а дъното на рекурсията връща друга стойност (няма смисъл да умножаваме по нула). По същия начин, по който „отделихме“ правилата за обхождане term и next като допълнителни аргументи на функциите sum/product, така можем да изнесем и правилото за изчисляване на резултата, заедно с нулевата си стойност, като нови два аргумента. Накрая получаваме функция от по-висок ред с 6 аргумента – двата краища на интервала, двете правила за обхождане, както и приложената операция с нейния неутрален елемент – винаги 0 за събиране, 1 за умножение. Тази функция е т.нар. **accumulate**, с която ще решаваме множество на пръв поглед различни задачи:

```
(define (accumulate op null-value term a next b)
      (if (> a b) null-value
         (op (term a) (accumulate op null-value term (next a) next b))))
```

Начинът на използване на accumulate е същият – трябва да измислим подходящите функции за обхождане, както и правилото за получаване на резултата (представлява процедура, най-често + или *), след което да извикаме еднократно accumulate с тях като аргументи. В рекурсивното извикване на accumulate променяме само a на (next a), останалите аргументи предаваме дословно. Горните три примера се решават чрез accumulate така:

```
(define (sum-ints a b)
      (define (id x) x)
      (define (1+ x) (+ x 1))
      (accumulate + 0 id a 1+ b))
```

```
(define (sum-cubes a b)
      (define (cube x) (expt x 3))
      (define (2+ x) (+ x 2))
      (accumulate + 0 cube a 2+ b)
)

(define (product-ints a b)
      (define (id x) x)
      (define (1+ x) (+ x 1))
      (accumulate * 1 id a 1+ b)
)
```

Възможно е да дефинираме наново обобщените функции за сума и произведение като частни случаи на accumulate, дори да не ги ползваме никога повече (те имат само образователна цел за нас):

```
(define (sum term a next b) (accumulate + 0 term a next b))
(define (product term a next b) (accumulate * 1 term a next b))
```

Нужно е всеки път да посочваме изрично null-value за комбиниращата операцията, защото няма как интерпретаторът да знае неутралните стойности за всяка една операция. Ако ще ползваме само събиране и умножение, можем да променим дефиницията на accumulate, но след няколко упражнения ще видим примери, при които ще използваме дефинирани от нас комбиниращи операции, за които трябва да указваме на accumulate какво да вземе за неутрална стойност. Операцията op трябва задължително да е двуаргументна – accumulate не може да прецени каква функция подаваме и просто разчита да бъде такава, за да върне коректен резултат. В повечето задачи ще използваме или събиране, или умножение, но не са изключени булевите функции за конюнкция и дизюнкция, както и други.

Не е задължително да се помни точно кода на функцията accumulate, но след като човек разбере как работи, не е трудно да го възпроизведе. Ключовите моменти са:

- обхождаме интервала чрез рекурсивно извикане за негов подинтервал с „преместено“ начало – това преместване се реализира от функцията next
- при достигане на края на интервала връщаме неутрална стойност, за да не повлияем на резултата от изчислението
- на всяка стъпка намираме общия член на сумата/произведението с функцията term за началото на интервала (оттам (term a))и го комбинираме с подходящата операция към резултата за следващата част от интервала

Така дефинираната функция accumulate е аналогична на операцията foldl, само че за числов интервал, вместо зададен списък от числа/данни. Тъй като рекурсивното извикване е просто и единично (в края на краищата реализираме само един цикъл), можем да напишем и линейно-итеративен вариант:

```
(define (accum-iter op null-value term a next b)
      (define (helper curr res)
            (if (> curr b) res
                (helper (next curr) (op (term curr) res))))
      (helper a null-value))
```

Накратко, този вариант работи по следния начин – променливата curr започва да обхожда интервала от a към b на стъпки, указани от next функцията (затова извикваме (next curr)), и на всяка итерация насъбира резултата с подходящата операция в res. Тази променлива пък е инициализирана с null-value, за да не променя крайния резултат.За повечето задачи, които ще решаваме, двете дефиниции са еквивалентни и ще връщат един и същ резултат. Разликата между двата метода е, че при първия крайният резултат се изчислява във вида (1+(2+(3+0))), а при втория – (3+(2+(1+0))). При стандартните събиране и умножение разлика няма, но може да се получи когато операцията op е некомутативна или работи с по-особен тип данни напр. списък от числа.

Остава само да се запознаем с малкия брат на accumulate, кръстен filter-accumulate. Тази функция обхожда интервала от числа по същия начин, само че на всяка стъпка проверява дали левият му край изпълнява някакво условие и едва тогава добавя общия член към резултата, а иначе преминава на следващата итерация. Новото „условие“ е едноаргументна числова функция-предикат, която също трябва да изберем и предадем коректно на filter-accumulate. Кодът на функцията изглежда по следния начин:

```
(define (filter-accum pred? op null-value term a next b)
      (cond [(> a b) null-value]
            [(pred? a) (op (term a)
                           (filter-accum pred op null-value
                                         term (next a) next b))]
            [else          (filter-accum pred op null-value
                                         term (next a) next b)]))
```

Разликите между втората и третата клауза е добавянето на общия член за текущата итерация към резултата само в случая, когато предикатът е изпълнен. Рекурсивното извикване е едно и също и подобно на това в стандартния accumulate – вместо a ползваме (next a). Това значи, че веднага можем да напишем по аналогичен начин и итеративен вариант на филтриращата функция, който ще изглежда така:

```
(define (filter-iter pred? op null-value term a next b)
      (define (helper curr res)
            (cond [(> curr b) res]
                  [(pred? a) (helper (next curr) (op (term curr) res))]
                  [else      (helper (next curr) res))]
      )
      (helper a null-value))
```

След като сме дефинирали четири варианта на функцията от по-висок ред accumulate, можем още по-добре да я анализираме и разберем как работи. Веднъж човек да вникне в нея, не е трудно да съобрази как да напише сам която и да е от разновидностите ѝ.

3. Задачи за самостоятелна подготовка – рекурсивни процедури и функции от по-висок ред

За следващите задачи използвайте **едно** подходящо извикване на функцията accumulate или filter-accumulate, както в примерните задачи. Във всяка задача става въпрос за **целочислен** интервал [a;b] и търсене само на цели числа.

1. Да се дефинира процедура, която намира броя на всички числа-палиндроми в даден интервал [a;b].

2. Да се дефинира процедура, която намира броя на всички делители на число в даден интервал [a;b].

3. Да се дефинира процедура, която проверява дали едно число е просто.

   Упътване: тъй като интерпретаторите не позволяват директното използване на булевите операции and и or като аргументи на accumulate, дефинирайте и предайте свои `(define (and2 x y) (and x y))` или `(define (or2 x y) (or x y))` като аргумента **op**. Помислете кои ще са неутралните стойности и какво ще връща функцията term в различните случаи.

4. Да се дефинира процедура, която проверява дали има прости числа в даден интервал [a;b].

5. Да се дефинира процедура, която проверява дали едно число е съвършено, т.е. дали сборът на всичките му делители (без самото число) е равен на числото.
```
(perfect? 6) → #t       ; 1+2+3=6
(perfect? 28) → #t      ; 1+2+4+7+14=28
(perfect? 496) → #t     ; 1+2+4+8+16+31+62+124+248=496
```

6. Да се дефинира процедура, която приема функция f и число n и намира стойността на сумата f(0)+f(1)+…+f(n).

7. Да се дефинира процедура, която приема функция f и намира броя на всички нейни неподвижни точки в даден интервал [a;b] (броят на всички цели x, за които f(x)=x).
```
(define (f x) (* x x))
(statPts f -5 5) → 2    ; {0;1}
```

8. Да се дефинира функция, която изчислява биномния коефициент C_k^n за дадени n и k.
```
(binomial 5 3) → 10
(binomial 49 6) → 13983816
```

9. Да се дефинира процедура, която проверява дали дадена функция е константна в даден интервал [a;b].
```
(define (f x) (+ (* x x) (* -5 x) 6))
(constant? f 2 3) → #t
(constant? f 2 4) → #f
```

10. Да се дефинира процедура, която намира стойността на sinx, използвайки развитието на функцията в ред на Тейлър (първите 10 члена са достатъчни за точност)
```
(sin2 0) → 0
(sin2 (/ pi 6)) → 0,49999999999999994
(sin2 (/ pi 2)) → 1,0
(sin2 pi) → 1,0348166767926159e-011    ;≈0,00000000001
```

    P.S. И факториелът, и степента x^k, нужни за общия член на реда, могат да се изчислят чрез accumulate, но не е това целта на задачата. Който иска, може да го направи за допълнително елементарно упражнение. Получените стойности могат да се различават в зависимост от използваните среди за разработка и интерпретатори.

4. Ламбда-функции. Функциите като обекти. Връщане на функции като резултат от други функции.

В този раздел ще покажа едно от най-известните и най-често използвани свойства на езика Scheme, както и на целия стил Функционално програмиране – анонимните „ламбда“ функции, които позволяват да дефинираме временни функции, които да използваме само веднъж, както и да връщаме функции като резултат от други функции. Както (може би) помним от първото упражнение, всяка дефинирана от нас функция представлява име, свързано с т.нар. процедурен обект. Този процедурен обект представлява наредена двойка от два указателя: през първия се съхранява информация за функцията като брой и имена на формалните параметри, както и кода на тялото ѝ, а вторият указва средата, в която е дефинирана тази функция, най-често глобалната. Ламбда конструкцията създава точно такива процедурни обекти, с които можем да боравим свободно така, както с всички останали променливи – можем да ги предаваме като аргументи на други функции или да ги взимаме като върнат резултат, можем дори и да ги извикваме със свои аргументи. Синтаксисът на една ламбда-функция е следният:

```
(lambda (<формални_параметри>) <тяло_на_функцията>)
(lambda (x) (+ x 1))
(lambda (x y) (- (expt x y) (expt y x)))
```

Формалните параметри са условните имена, с които се взимат аргументите на функцията (както при дефиниция с define), а тялото може да бъде една или повече команди по същия начин, както в тялото на define. Параметрите могат да бъдат произволен брой, дори 0, стига само имената им да са разделени с празни места в списъка. Разликата между двата типа дефиниции е, че define веднага свързва име към процедурния обект, докато с lambda той остава „висящ“ във въздуха. Какво ни пречи тогава да го свържем с някакво име така, както дефинираме (define x 5)?

```
(define f (lambda (x) (* x x)))
```

По този начин указваме, че този процедурен обект ще ползва един аргумент, към който ще се обръща с името x, и ще връща неговия квадрат. lambda специалната форма е полезна точно когато искаме да „изолираме“ името на една функция (без списъка с аргументите ѝ), за да предадем това име на други функции. Тази дефиниция е равносилна с:

```
(define (f x) (* x x))
```

Използвайки модела на средите, можем да видим, че двете дефиниции създават една и съща картинка. Забележете еднаквото тяло на функциите в двата случая. Нека сме дефинирали f чрез специалната форма lambda. Ако поискаме да видим просто стойността на f, ще разберем, че това е процедура. За да получим стойност за някакво x, трябва да я извикаме по същия начин, по който извикваме всяка друга функция: отваряме скоба, поставяме името ѝ на първо място, след което изброяваме параметрите и затваряме скобата, а именно:

```
(f 5)
> 25
```

Още една прилика с дефинираните с define функции. Ламбда-функциите имат три най-често срещани употреби:

- като временни функции, които се предават веднъж на функция от по-висок ред и не се използват повече
- като стойност, върната от друга функция
- като обикновени процедури, извикани със своите аргументи

Ще се срещаме с тях най-често чрез първия метод. Кога ни се е налагало да дефинираме някакви прости функции, след което да ги използваме еднократно като аргумент на функция от по-висок ред? Единственият път, когато въобще сме работили с функции от по-висок ред, при функцията accumulate и нейните производни. Използвайки еквивалентността на define и lambda (що се отнася до функции), имаме избор между две различни, но еквивалентни дефиниции:

```
(define (sum-ints a b)
      (define (id x) x)
      (define (1+ x) (+ x 1))
      (accumulate + 0 id a 1+ b))

(define (sum-ints a b)
      (accumulate +
                  0
                  (lambda (x) x)
                  a
                  (lambda (x) (+ x 1))
                  b))
```

Орязахме тялото на функцията ни до една команда – единственото извикане на accumulate, без никакви допълнителни дефиниции, но пък се получи по-дълъг и неугледен ред код (само заради това е разделен на няколко реда, няма значение иначе). При по-сложни функции temp и next би било още по-неудобно да използваме ламбда-функции, но решението би било все така вярно. Дали ще изберем вложени дефиниции или анонимни функции, или комбинация от двата метода според случая, е личен избор на програмиста и, стига решението да е вярно, по никакъв начин няма да се отрази на оценката на което и да е контролно или изпит. Който желае, може да напише някои от задачите за самостоятелна подготовка, използвайки ламбда-функции за свикване със синтаксиса.

Нека имаме следната задача: да се дефинира процедура, която приема една функция f и връща друга функция, еквивалентна на f(x)+2. Съществото е тази процедура да има само един параметър – първоначалната функция и нищо повече. Решението ще изглежда така:

```
(define (move f) (lambda (x) (+ (f x) 2)))
```

Всички условия са изпълнени точно: взимаме функция като аргумент и връщаме функция като резултат, дефинирана с lambda. Ще напишем и анализираме следните няколко реда код:

```
(define (f x) (* 3 x))
(define g (move f))      ;може и (move (lambda (x) (* 3 x)))
(g 5) → 17
((move f) 5) → 17
((move (lambda (x) (* 3 x))) 5) → 17
```

Взехме произволна функция f и дефинирахме с g нейната изместена. Забележете съответствието между тялото на f и тялото на анонимната функция в алтернативната дефиниция на g. Тъй като move ще е върнало процедура, то и g става процедура, а не просто променлива, заради

което се извиква с параметър 5 като обикновена функция. Ако искахме все пак да дефинираме g както сме свикнали да дефинираме други функции, трябва да напишем това:

```
(define (g x) ((move f) x))
```

Както „ограждаме“ g в скоби, показвайки, че е процедура, така се налага и да „оградим“ израза (move f) в скоби, за да го използваме като самостоятелна процедура. Последният начин на употреба на ламбда-функциите е най-банален и най-рядко срещан. Казахме, че специалната форма ламбда връща процедурен обект, който можем да използваме като всеки друг. В такъв случай имаме следното съответствие:

```
(define f (lambda (x) (+ x 1)))
(f 5) → 6

((lambda (x) (+ x 1)) 5) → 6
```

При извикването чрез ламбда-функцията създаваме един временен **x**, на който след това присвояваме стойността 5, за да изчислим x+1. Обърнете внимание как при това извикване просто преписахме стойността на f (целият ламбда-израз) и ѝ предадохме същите параметри. Такива локални дефиниции са полезни в много случаи и могат да заместят вложените дефиниции в по-сложни функции. Решението с ламбда функция при повече такива локални дефиниции обаче може да не е най-уместното, а дефиниране на временни функции би било още по-неудобно и грозно:

```
((lambda (f a b) (+ (f a) (f b))) (lambda (x) (expt 2 x)) 3 5) → 40
```

В един ред дефинирахме анонимно и извикахме функция от по-висок ред, която смята f(a)+f(b) за функцията $f(x)=2^x$, a=3 и b=5 ($2^3+2^5=40$). Има по-удобна конструкция за локални дефиниции, която има и особен метод на оценяване, за това за нея е отделена отделна тема.

5. Локални дефиниции. Специални форми let и let*. Оценяване на изрази с let и let*.

По метода на оценяване чрез модела на средите всяка извикана функция създава своя среда-разширение, където дефинира формалните си параметри, преди да ги използва за изчисление. Същото важи и за ламбда-фунцкията от края на предишния раздел. Тази конструкция, позволяваща краткотрайни локални дефиниции, се използва често и затова е създадена специалната форма let със следния синтаксис:

```
(let [(<пром.1> <израз1>)
      (<пром.2> <израз2>)
                ...
      (<пром.N> <изразN>)]
  <тяло>)
```

Подобно на синтаксиса при специалната форма lambda, така и тук локалните дефиниции са отделени една от друга в скоби, но всички дефиниции са събрани в списък, т.е. заградени от външен чифт скоби. let-изразът, еквивалентен на последната дефиниция с ламбда-функция, ще изглежда така:

```
(let [(f (lambda (x) (expt 2 x))
      (a 3)
      (b 5)]
   (+ (f a) (f b))
→ 40
```

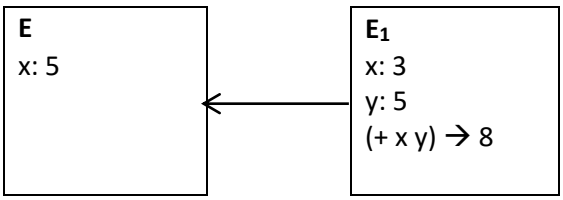
Реално погледнато let не внася нова функционалност в езика. Както и много други специални форми в Scheme, така и let може да се замени от ламбда конструкция, но служи като „синтактична захар“. При подаването на такъв израз на интерпретатора, веднага ще получим резултата от изчислението. Внимавайте: при дефиниция на локални функции е допустима само ламбда-конструкция като написаната в примера, а не такава със синтаксиса на define. Идва най-важната част в тази тема – начинът на оценяване на let-изрази. Нека имаме същия израз, който сме извикали в глобалната среда **E**. let създава среда-разширение, която ще кръстим E_1, след което извършва следната последователност от действия:

- дефинира в E_1 имена на локалните променливи - f, a и b
- **оценява в E** съответните им изрази и присвоява стойностите им на новите имена в E_1
- оценява тялото **според дефинициите в E_1**

Оценяването на <израз1>, ... , <изразN> става паралелно и независимо един от друг. При горния пример подробността къде се оценяват няма значение, заради което ще покажа по-подходящ пример:

```
(define x 5)
(let [(x 3)
      (y x)]
   (+ x y))
→ ?
```

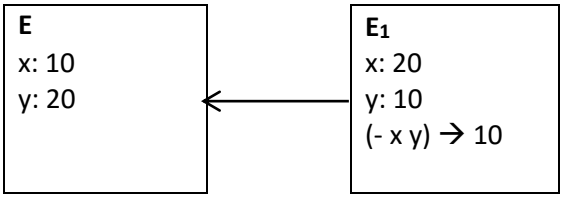
Проста конструкция, която поставя въпроса: каква е стойността на **y**? Нека минем през оценяването стъпка по стъпка. В глобалната среда имаме име **x** и стойност числото 5. let създава среда-разширение E1 и записва в нея новото име **x** със стойност 3. Подобно на scope-принципа в C++, **x** ще има стойност 3 навсякъде в тялото на let-функцията, а извън нея – 5. Стойността на **y** ще се търси **само** в глобалната среда (където сме извикали този let-израз). Ако в нея има име **x**, то **y** ще „препише“ неговата стойност в средата-разширение. Ако обаче в средата-майка няма име **x**, ще се получи грешка и интерпретаторът няма да върне нищо. Следователно **y** ще е равен на 5 и (+ x y) ще се изчисли като (+ 3 5) за краен резултат 8.



Друг класически пример е следният:

```
(define x 10)
(define y 20)
(let [(x y)
      (y x)]
   (- x y))
→ ?
```

Създаваме x и y в глобалната среда със стойности 10 и 20. let-конструкцията създава среда разширение, в която дефинира нови, локални x и y, като им присвоява стойностите от глобалната среда. Локалното y не вижда локалния x и не може да се влияе от него, както и обратното. Моделът на средите при такава дефиниция ще изглежда така:



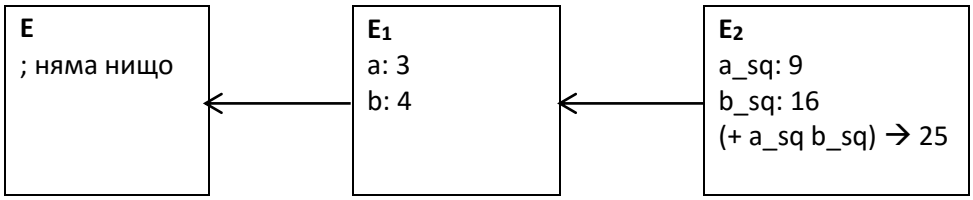
Вижда се ограничението за независещи една от друга локални дефиниции и човек би се запитал: има ли как да го заобиколим и какви специални форми ще използваме? lambda? let? Нещо съвсем различно? И трите начина са възможни, но ще демонстрирам само последните два, първият е непрактичен.

Позволено е let-изрази да бъдат влагани, т.е. тялото на един израз да бъде цял друг let-израз. Ще напишем let-конструкция, която дефинира локално a=3 и b=4, след което смята a^2+b^2, дефинирайки също a^2 и b^2 локално. Следната дефиниция ще е некоректна:

```
(let [(a 3)
      (b 4)
      (a_sq (* a a))
      (b_sq (* b b))]
   (+ a_sq b_sq))
→ Грешка!
```

В глобалната среда, където извикваме let-израза, нямаме дефинирани a и b и тогава a_sq и b_sq остават недефинирани. Трябва да използваме вложени let-изрази:

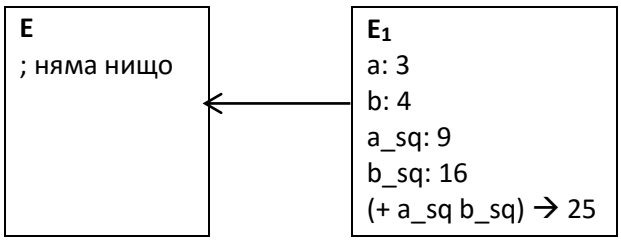
```
(let [(a 3)
      (b 4)]
   (let [(a_sq (* a a))
         (b_sq (* b b))]
      (+ a_sq b_sq)))
→ 25
```



Тъй като вътрешният let е извикан в средата E_1 (там се оценява тялото на външния let), то E_2 се създава като нейно разширение и a_sq и b_sq се оценяват според дефинииите в E_1, а не в E. Такъв тип израз би бил неудобен, ако имаме повече зависимости между локалните дефиниции, което налага нуждата от нова специална форма, кръстена let*. Синтаксисът е абсолютно същият, като единствената разлика е в начина на оценяване. Всяка локална дефиниция „вижда“ тези пред нея и може да ги използва, т.е. оценяването на <израз1>, … , <изразN> се извършва първо в новосъздадената среда-разширение и след това (ако не се намерят подходящите имена) се преминава нагоре към средата-майка. Това означава, че при let* подредбата на дефинициите има значение, за разлика от обикновения let. Заради различното оценяване примери като предишните два ще върнат различен резултат при let и let*.

Това са най-важните неща около двете специални форми за локални дефиниции. Запомнете разликите между тях и няма да имате проблеми да боравите и с двете. Накрая остава да покажа как се решава задачата с let* и картинката на средите:

```
(let* [(a 3)
       (b 4)
       (a_sq (* a a))
       (b_sq (* b b))]              ; не забравяйте скобите, обединяващи
   (+ a_sq b_sq)                    ; отделните дефиниции в списък
) → 25
```



6. Бонус за любознателните

Съществува един вид „припокриване“ между филтриращата и нефилтриращата функция, което си проличава в по-прости примери, като за намиране на сумата на всички четни числа в даден интервал. Можем да използваме директно filter-accumulate или обикновения accumulate с по-сложно зададени стъпка и начало на цикъла. Ето двете решения:

```
(define (sum-evens1 a b)
      (define (my-even? x) (= (remainder x 2) 0)) ;можем да ползваме и
                                                    вградения предикат even?
      (define (id x) x)
      (define (1+ x) (+ x 1))
      (filter-accum my-even? + 0 id a 1+ b))

(define (sum-evens2 a b)
      (define (id x) x)
      (define (2+ x) (+ x 2))
      (accumulate + 0 id (if (even? a) a (+ a 1)) 2+ b))
```

Когато премахваме филтъра трябва да внимаваме откъде започваме обхождането – иначе ще съберем всички числа със същата четност като a (ами ако a е нечетно?). Задачата има и трето решение:

```
(define (sum-evens3 a b)
      (define (term x) (if (even? x) x 0))
      (define (1+ x) (+ x 1))
      (accumulate + 0 term a 1+ b))
```

Тук проверката се извършва във функцията за общ член, която вместо неподходящите нечетни числа подава нули (т.е. неутралния елемент) на операцията събиране. Този подход също не е универсален и грешни резултати се получават именно при по-различна операция op (разбирайте, когато изграждаме списъци от числа). При други задачи пък измислянето на подходяща стъпка, която да действа като филтър, може да е по-трудно. Нека имаме задачата за сбор на всички прости числа в даден интервал, като използваме предиката *prime?* наготово. Решението, което се очаква веднага да можем да напишем, е чрез бавно и последователно обхождане на всички числа, като ги филтрираме с prime?:

```
(define (sum-primes a b)
      (define (id x) x)  ; след като намерим някое просто число, го
                           добавяме без да го изменяме
      (define (1+ x) (+ x 1)) ; имена като 1+ и id са „стандартни“, но
                                далеч не задължителни
      (filter-accum prime? + 0 id a 1+ b))
```

Ако е поставено условие да не използваме filter-accumulate, тогава трябва да измислим функция step такава, че да връща следващото просто число за следващата итерация.

```
(define (sum-primes* a b)
      (define (id x) x)
      (define (nextPrime x) (if (prime? (+ x 1)) (+ x 1)
                                                 (nextPrime (+ x 1))))
      (accumulate + 0 id (if (prime? a) a (nextPrime a)) nextPrime b))
```

В тялото на nextPrime проверяваме винаги x+1, а не x, защото искаме тя да връща следващото по големина просто число, в противен случай ще зацикли безкрайно при достигане на първото такова. Вместо да извикаме accumulate за интервала, подаден на нашата sum-primes*, проверяваме първо дали левият край е просто число и в противен случай започваме от следващото. Обхождането на интервала става на „скокове“ от едно просто число на друго, осъществени от nextPrime функцията.

Край.